Digital Systems Design with VHDL

Instructors
Dr. Amany Sarhan
Dr.Mahmoud Alshewimy
2014/2015

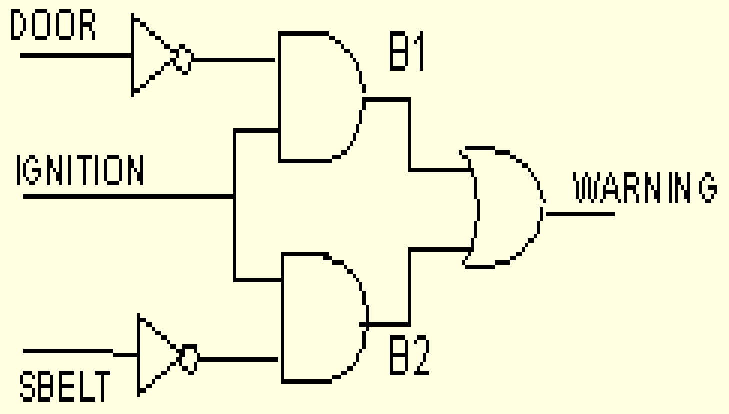
Definition

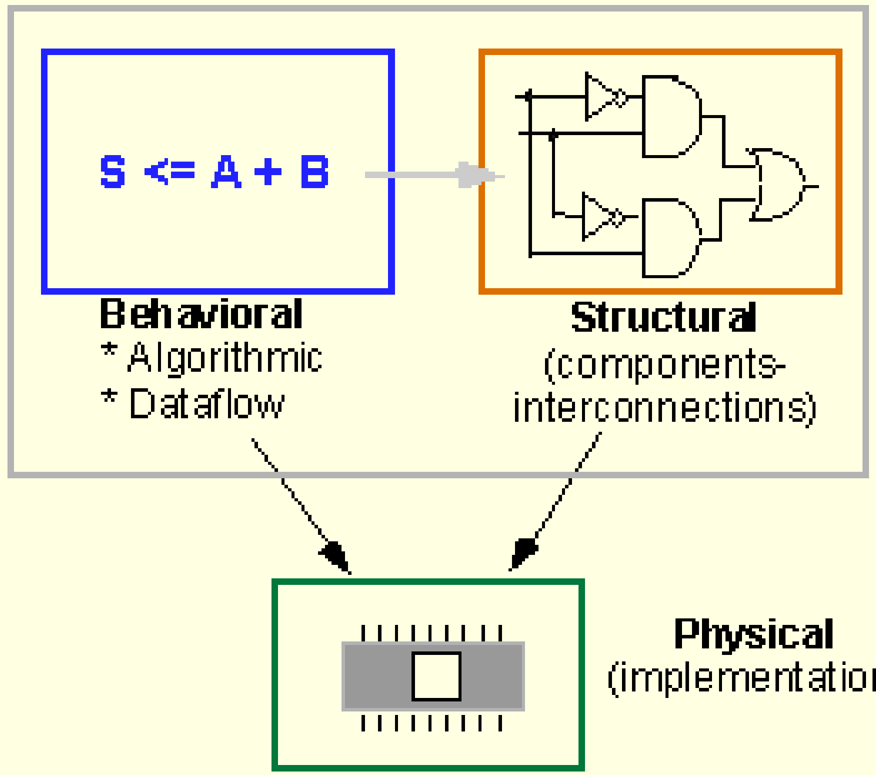
- VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.
- It is a hardware description language with the goal to develop very high-speed integrated circuits.
- It has become now one of industry’s standard languages used to describe digital systems.
- The other widely used hardware description language is Verilog. Both are powerful languages that allow you to describe and simulate complex digital systems.

Introduction

- VHDL can be used to model digital circuits.
- Having a model of the circuit allows for simulation and testing of the design for proper operation.
- But maybe more importantly, the act of creating the model from VHDL code is a valuable and interesting learning experience in itself.
- Second, VHDL and other hardware description languages are used as one of the first steps in creating large digital integrated circuits.
- The VHDL code is used to magically create digital circuits in a process known as synthesis.

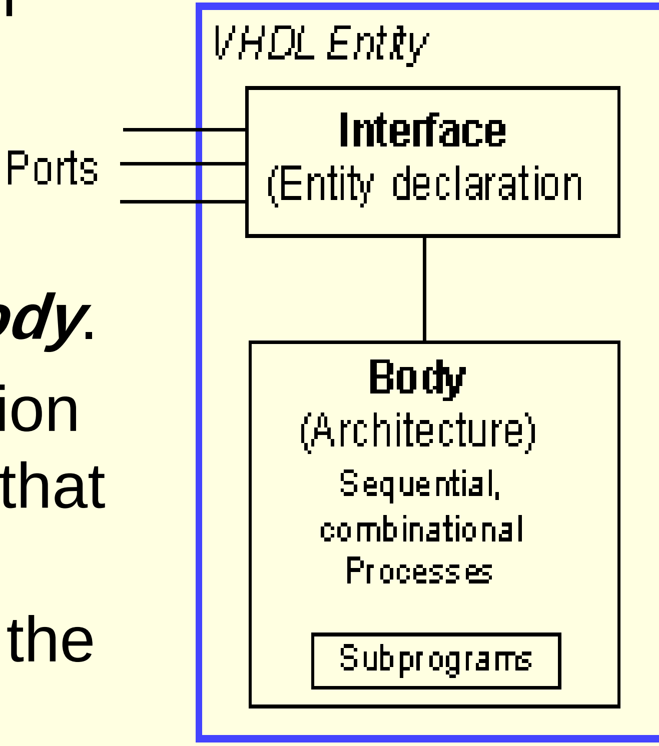
Levels of representation and abstraction





Basic Structure of a VHDL file

- A design entity that can contain other entities that are then considered components of the top-level entity.
- Each entity is modeled by an ***entity declaration*** and an ***architecture body***.
- One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently

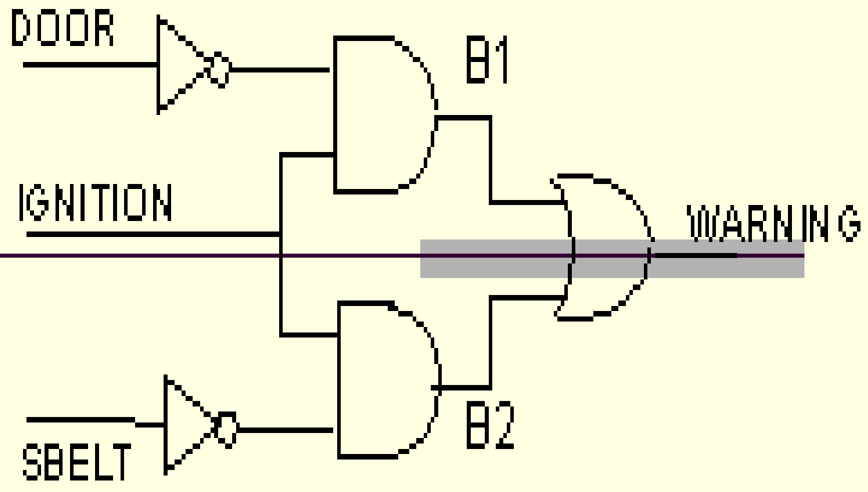


VHDL Statements

1- The Entity

- The *entity* is VHDL's version of the black box.
- The VHDL entity construct provides a method to abstract the functionality of a circuit description to a higher level.
- It describes how the black box interfaces with the outside world.
- Since VHDL is describing a digital circuit, the entity simply lists the various input and outputs to the underlying circuitry.

```
entity   entity_name    is
  [port_clause]
end   entity_name;
```



```
port (
   port_name : mode data_type;
   port_name : mode data_type;
   port_name : mode data_type
     );
```

Example

```
entity    BUZZER    is
    port (DOOR, IGNITION, SBELT: in std_logic;
            WARNING: out std_logic);
 end    BUZZER;
```

```
entity my_4t1_mux is
   port ( D3,D2,D1,D0 : in std_logic;
            SEL : in std_logic_vector(1 downto 0);
            MX_OUT : out std_logic);
end my_4t1_mux;
```

```
entity dff_sr is
   port (D,CLK,S,R: in std_logic; Q,Qnot: out std_logic);
        end dff_sr;
```

```
entity my_nand3 is
   port ( A,B,C : in std_logic;
            F : out std_logic);
end my_nand3;
```

```
entity mux4_to_1 is
    port (I0,I1,I2,I3: in std_logic_vector(7 downto 0);
               SEL : in std_logic_vector(1 downto 0);
                      OUT1: out std_logic_vector(7 downto 0));
       end mux4_to_1;
```

Architecture body

The architecture body specifies how the circuit operates and how it is implemented.

An entity or circuit can be specified in a variety of ways:

1. behavioral
2. structural (interconnected components)
3. A combination of the above

1. The behavioral description

```
architecture behavioral of BUZZER is
    begin
        WARNING <= (not DOOR and IGNITION) or (not SBELT
        and IGNITION);
    end behavioral;
```

<u>Example 1:</u> The behavioral description of a two-input AND gate is shown below.

```
entity AND2 is
      port (in1, in2: in std_logic;
                  out1: out std_logic);
end AND2;

architecture behavioral_2 of AND2 is
  begin
              out1 <= in1 and in2;
end behavioral_2;
```

<u>Example 2:</u> An example of a two-input XNOR gate is shown below.

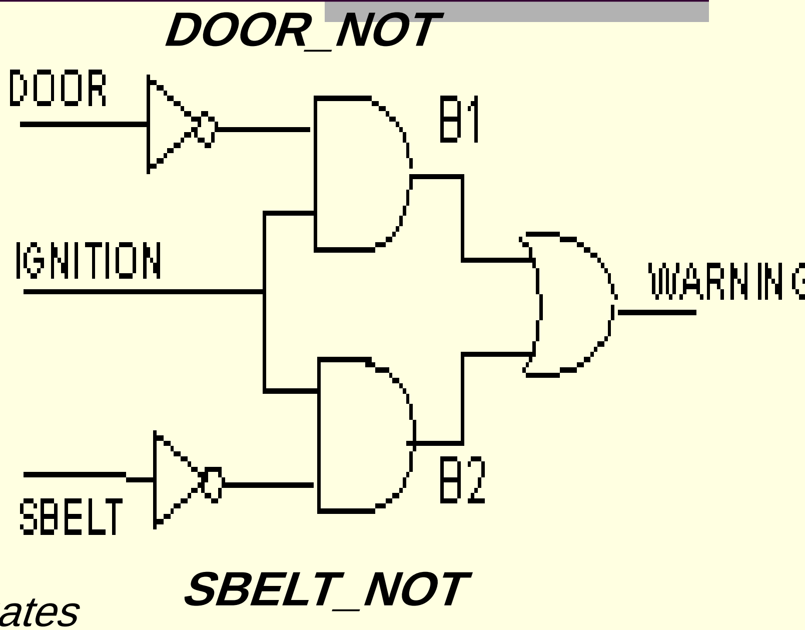
```
    entity XNOR2 is
        port (A, B: in std_logic;
                       Z: out std_logic);
         end XNOR2;

    architecture behavioral_xnor of XNOR2 is
         -- signal declaration (of internal signals X, Y)
         signal X, Y: std_logic;
begin
          X <= A and B;
          Y <= (not A) and (not B);
          Z <= X or Y;
         end behavioral_xnor;
```

2- Structural description

- ***architecture name of entity_name is***
- *-- Declarations of used components*
- ***component Comp1_name***
- ***port (xxxx: in ; xxxx: out);***
- ***end Comp1_name;***
- ***-----***
- ***-----***
- ***begin***
- *-- Declarations of variables and signals*
- ***------***
- *-----Component instantiations statement*
- ***PUT here instants of the declared components above as***
- ***U1: Comp1_name port map (inputs, outputs)***
- ***As from the logic diagram of the circuit***

2- Structural description

```
architecture structural of BUZZER is
        -- Declarations
         component AND2
                  port (in1, in2: in std_logic;
                          out1: out std_logic);
         end component;
        component OR2
                 port (in1, in2: in std_logic;
                          out1: out std_logic);
        end component;
        component NOT1
                 port (in1: in std_logic;
                         out1: out std_logic);
        end component;
----declaration of signals used to interconnect gates
signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;
   begin
-----Component instantiations statements
        U0: NOT1 port map (DOOR, DOOR_NOT);
        U1: NOT1 port map (SBELT, SBELT_NOT);
        U2: AND2 port map (IGNITION, DOOR_NOT, B1);
       U3: AND2 port map (IGNITION, SBELT_NOT, B2);
       U4: OR2  port map (B1, B2, WARNING);
   end structural;
```



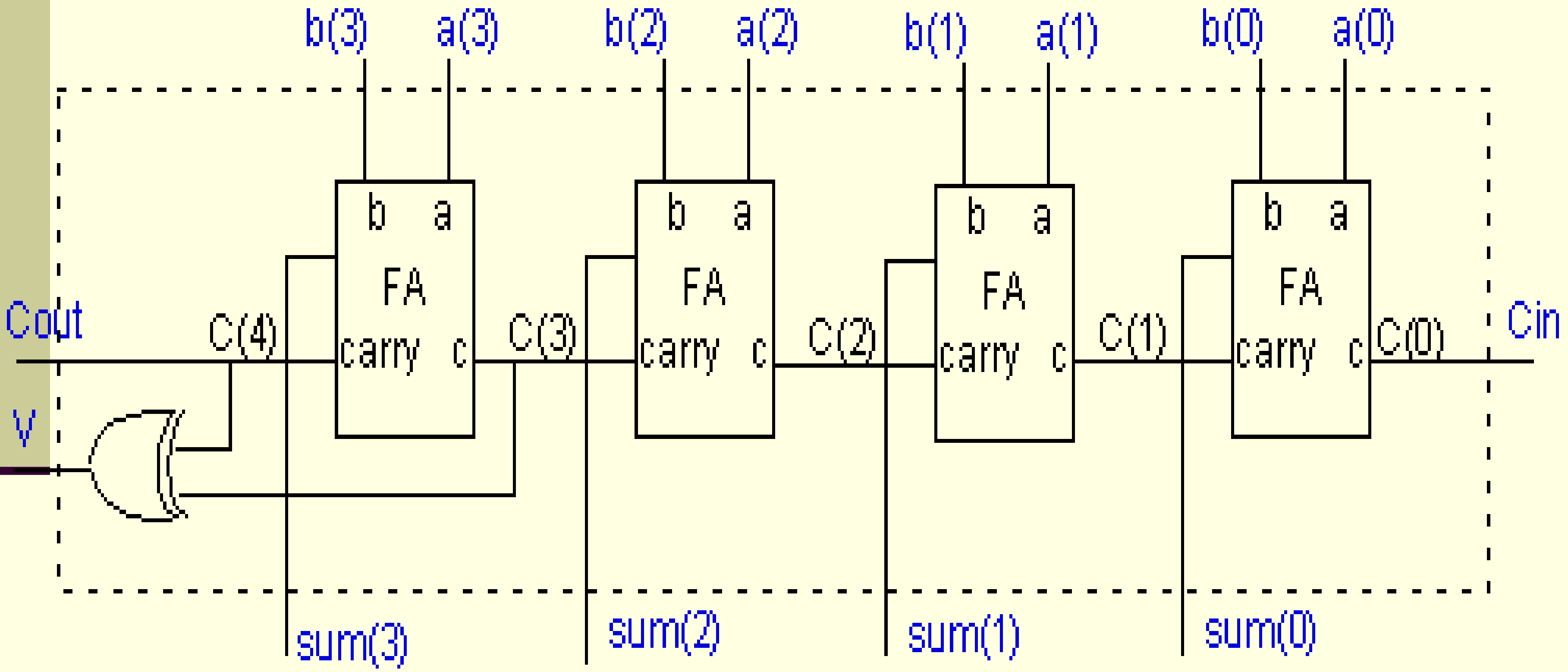
Another version:

```
U0: NOT1 port map (in1 => DOOR, out1 => DOOR_NOT);
U1: NOT1 port map (in1 => SBELT, out1 => SBELT_NOT);
U2: AND2 port map (in1 => IGNITION, in2 => DOOR_NOT,
    out1 => B1);
U3: AND2 port map (in1 => IGNITION, in2 => SBELT_NOT,
    B2);
U4: OR2  port map (in1 => B1, in2 => B2, out1 =>
    WARNING);
```

Hierarchal description

- Structural modeling of design is used in hierarchical design, in which one can define components of units that are used over and over again.
- Once these components are defined they can be used as blocks, cells or macros in a higher level entity.
- This can significantly reduce the complexity of large designs.
- Hierarchical design approaches are always preferred over flat designs.
- We will illustrate the use of a hierarchical design approach for a 4-bit adder

$$\text{sum} = (A \oplus B) \oplus C$$
$$\text{carry} = AB + C(A \oplus B)$$



1 – Define the full adder entity

```
library  ieee;
use  ieee.std_logic_1164.all;
-- definition of a full adder
      entity FULLADDER is
    port (a, b, c: in std_logic;
                     sum, carry: out std_logic);
      end FULLADDER;
```

2- define the behavior of the full adder entity

```
 architecture fulladder_behav of FULLADDER is
begin
sum <= (a xor b) xor c ;
carry <= (a and b) or (c and (a xor b));
     end fulladder_behav;
```

3- Define 4-bit full adder entity

```
library  ieee;
use  ieee.std_logic_1164.all;
      entity FOURBITADD is
      port (a, b: in std_logic_vector(3 downto 0);
               Cin : in std_logic;
                           sum: out std_logic_vector (3 downto
0);
                           Cout, V: out std_logic);
      end FOURBITADD;
```

4- define the structure of the 4-bit full adder entity

```
      architecture fouradder_structure of FOURBITADD is
           signal c: std_logic_vector (4 downto 0);

      component FULLADDER
                port(a, b, c: in std_logic;
                       sum, carry: out std_logic);
      end component;
      begin
      FA0: FULLADDER port map (a(0), b(0), Cin, sum(0), c(1));
      FA1: FULLADDER port map (a(1), b(1), C(1), sum(1), c(2));
```

```
FA2: FULLADDER port map (a(2), b(2), C(2), sum(2), c(3));
FA3: FULLADDER port map (a(3), b(3), C(3), sum(3), c(4));

   V <= c(3) xor c(4);
   Cout <= c(4);

end fouradder_structure;
```

Library and Packages:

- A library can be considered as a place where the compiler stores information about a design project.
- A VHDL package is a file or module that contains declarations of commonly used objects, data type, component declarations, signal, procedures and functions that can be shared among different VHDL models.
- We mentioned earlier that **std_logic** is defined in the package **ieee.std_logic_1164** in the ieee library.
- In order to use the std_logic one needs to specify the library and package.
- This is done at the beginning of the VHDL file using the library and the use keywords as follows:

```
library  ieee;
use  ieee.std_logic_1164.all;
```

- The ***.all*** extension indicates to use all of the ieee.std_logic_1164 package.

- The **Xilinx** Foundation Express comes with several packages.
- **<u>ieee Library:</u>**
- ***std_logic_1164 package:*** defines the standard datatypes
- ***std_logic_arith package:*** provides arithmetic, conversion and comparison functions for the signed, unsigned, integer, std_ulogic, std_logic and std_logic_vector types
- ***std_logic_unsigned***
- ***std_logic_misc package:*** defines supplemental types, subtypes, constants and functions for the std_logic_1164 package.
- To use any of these one must include the library and use clause:

  library ieee;

  use ieee.std_logic_1164.all;

  use ieee.std_logic_arith.all;

  use ieee.std_logic_unsigned.all;

Declaring new packages

- The **syntax to declare a package** is as follows:

```
-- Package declaration
 package name_of_package is
       package declarations
 end package name_of_package;

-- Package body declarations
     package body name_of_package is
       package body declarations
    end package body name_of_package;
```

Example:

- For instance, the basic functions of the AND2, OR2, NAND2, NOR2, XOR2, etc. components need to be defined before one can use them.
- This can be done in a package, e.g. ***basic_func*** for each of these components, as follows.

```
library  ieee, my_func;
use  ieee.std_logic_1164.all, my_func.basic_func.all;
-- Package declaration
library ieee;
use ieee.std_logic_1164.all;
package basic_func is

   -- AND2 declaration as component
   component AND2
      generic (DELAY: time :=5ns);
      port (in1, in2: in std_logic; out1: out std_logic);
   end component;

   -- OR2 declaration as component
   component OR2
      generic (DELAY: time :=5ns);
      port (in1, in2: in std_logic; out1: out std_logic);
   end component;

end package basic_func;
```

```
-- Package body declarations
library ieee;
use ieee.std_logic_1164.all;
package body basic_func is

    -- 2 input AND gate declaration and description

        entity AND2 is
            generic (DELAY: time);
            port (in1, in2: in std_logic; out1: out std_logic);
        end AND2;

        architecture model_conc of AND2 is
         begin
            out1 <= in1 and in2 after DELAY;
        end model_conc;
```

-- 2 input OR gate declaration and description

```
entity OR2 is
    generic (DELAY: time);
    port (in1, in2: in std_logic; out1: out std_logic);
end OR2;

architecture model_conc2 of OR2 is
    begin
        out1 <= in1 or in2 after DELAY;
end model_conc2;

end package body basic_func;
```

Lexical Elements of VHDL
1. Identifiers

- ***Identifiers*** are user-defined words used to name objects in VHDL models (for input and output signals, the name of a design entity and architecture body).
- When choosing an identifier one needs to follow these basic rules (**basic identifiers**):

1- May contain only alpha-numeric characters (A to Z, a to z, 0-9) and the underscore (_) character

2- The first character must be a letter and the last one cannot be an underscore.

3- An identifier cannot include two consecutive underscores.

4- An identifier is case insensitive (ex. And2 and AND2 or and2 refer to the same object)

5- An identifier can be of any length.

- **Examples of valid identifiers** are: X10, x_10, My_gate1.
- Some **invalid identifiers** are: _X10, my_gate@input, gate-input.

- **Extended identifier** have different rules which allow identifiers with any sequence of characters as follows:

1- An extended identifier is enclosed by the backslash, "\", character.

2- An extended identifier is case sensitive.

3- An extended identifier is different from reserved words (keywords) or any basic identifier (e.g. the identifier \identity\ is allowed)

4- Inside the two backslashes one can use any character in any order, except that a backslash as part of an extended identifier must be indicated by an additional backslash. As an example, to use the identifier BUS:\data, one writes: \BUS:\\data\

5- Extended identifiers are allowed in the VHDL-93 version but not in VHDL-87

- Some **examples of legal identifiers** are:
- Input, \Input\, \input#1\, \Rst\\as\

2 Keywords (Reserved words)

- These ***keywords cannot be used*** as identifiers for signals or objects we define.
- We have seen several of these reserved words already such as:

  in, out, or, and, port, map, end, etc.
- Keywords are often printed in ***boldface***, as is done in this book.
- Extended identifiers can make use of keywords since these are considered different words (e.g. the extended identifier ***lendl*** is allowed).

3 Numbers

- The default number representation is the **decimal system**.
- VHDL allows integer literals and real literals.
- Integer literals consist of whole numbers without a decimal point, while real literals always include a decimal point.
- Exponential notation is allowed using the letter “E” or “e”. For integer literals the exponent must always be positive. Examples are:
- **Integer literals**: 12   10   256E3   12e+6
- **Real literals**:   1.2   256.24 3.14E-2
- The number –12 is a combination of a negation operator and an integer literal.

- To express a number in a base different from the base “10”, one uses the following convention: ***base#number#.***
- A few examples follow.

Representing the decimal number “18”:

- **Base 2**:   2#10010#
- **Base 16**: 16#12#
- **Base 8**:   8#22#

Representing the decimal number “29”:

- **Base 2**:   2#11101#
- **Base 16**: 16#1D#
- **Base 8:**   8#35#

- To make the readability of large numbers easier, one can insert underscores in the numbers as long as the underscore is not used at the beginning or the end.
- 2#1001_1101_1100_0010#
- 215_123

4 Characters, Strings and Bit Strings

- To use a character literal in a VHDL code, one puts it in a single quotation mark, as: ***'a', 'B', ','***
- A string of characters are placed in double quotation marks as:

  "This is a string",

  "This is a """String""."
- Any printing character can be included inside a string.
- A bit-string represents a sequence of bit values. In order to indicate that this is a ***bit string***, one places the 'B' in front of the string: ***B"1001"***.
- One can also use strings in the hexagonal or octal base by using the X or O specifiers, respectively. Some examples are:

  Binary: B"1100_1001", b"1001011"

  Hexagonal: X"C9", X"4b"

  Octal: O"311", o"113"

Constants

- A ***constant*** can have a single value of a given type and **cannot be changed** during the simulation.
- A constant is declared as follows,

  constant list_of_name_of_constant: type [:= initial value] ;
- where the initial value is optional.
- Constants can be declared at the start of an architecture and can then be used anywhere within the architecture.
- Constants declared within a process can only be used inside that specific process.

Examples:

constant RISE_FALL_TME: time := 2 ns;

constant RISE_TIME, FALL_TIME: time:= 1 ns;

constant DATA_BUS: integer:= 16;

Variable

- A ***variable*** can have a single value, as with a constant, but a variable **can be updated** using a variable assignment statement.
- The variable is **updated without any delay** as soon as the statement is executed.
- Variables must be declared inside a process.
- The variable declaration is as follows:

variable list_of_variable_names: type [:= initial value] ;

A few examples follow:

```
variable CNTR_BIT: bit :=0;
variable VAR1: boolean :=FALSE;
variable SUM: integer range 0 to 256 :=16;
variable STS_BIT: bit_vector (7 downto 0);
```

- A variable can be updated using a variable assignment such as: ***Variable_name := expression;***

Signal

- Signals are declared with the following statement:

  signal list_of_signal_names: type [:= initial value] ;

- Some examples of signals are:

  signal SUM, CARRY: std_logic;
  signal CLOCK: bit;
  signal TRIGGER: integer :=0;
  signal DATA_BUS: bit_vector (0 to 7);
  signal VALUE: integer range 0 to 100;

- Signals are **updated** when their **signal assignment statement is executed**, after a certain delay, as illustrated below,

  SUM <= (A xor B) after 2 ns;

- The sum signal will have the value after 2 ns of computing A xor B.

difference between variables and signals

- It is important to understand the **difference between variables and signals**, particularly how it relates to when their value changes.
- A **variable changes instantaneously when the variable assignment is executed**.
- On the other hand, a **signal changes a delay after the assignment expression is evaluated**.
- If no delay is specified, the signal will change after a ***delta*** **delay**.
- This has important consequences for the updated values of variables and signals.

Example:

Example of a process using Variables

```
architecture VAR of EXAMPLE is
    signal TRIGGER, RESULT:
    integer := 0;
begin
    process
      variable variable1: integer :=1;
      variable variable2: integer :=2;
      variable variable3: integer :=3;
    begin
    wait on TRIGGER;
   variable1 := variable2;
   variable2 := variable1 + variable3;
   variable3 := variable2;
   RESULT <= variable1 + variable2
    + variable3;
    end process;
end VAR
```

Example of a process using Signals

```
 architecture SIGN of EXAMPLE is
     signal TRIGGER, RESULT:
     integer := 0;
     signal signal1: integer :=1;
     signal signal2: integer :=2;
     signal signal3: integer :=3;
 begin
     process
     begin
            wait on TRIGGER;
            signal1 <= signal2;
            signal2 <= signal1 + signal3;
            signal3 <= signal2;
     RESULT  <= signal1 + signal2 +
signal3;
     end process;
 end SIGN;
```

Data Types defined in the Standard Package

Types defined in the Package *Standard* of the *std* Library		
Type	**Range of values**	**Example**
bit	'0', '1'	signal A: bit :=1;
bit_vector	an array with each element of type bit	signal INBUS: bit_vector(7 downto 0);
boolean	FALSE, TRUE	variable TEST: Boolean :=FALSE'
character	any legal VHDL character (see package standard); printable characters must be placed between single quotes (e.g. '#')	variable VAL: character :='$';
file_open_kind*	read_mode, write_mode, append_mode	
file_open_status*	open_ok, status_error, name_error, mode_error	
integer	range is implementation dependent but includes at least $-(2^{31}-1)$ to $+(2^{31}-1)$	constant CONST1: integer :=129;
natural	integer starting with 0 up to the max specified in the implementation	variable VAR1: natural :=2;

Data Types defined in the Standard Package

positive	integer starting from 1 up the max specified in the implementation	variable VAR2: positive :=2;
real*	floating point number in the range of –1.0 x 10^{38} to +1.0x 10^{38} (can be implementation dependent. *Not supported by the Foundation synthesis program.*	variable VAR3: real :=+64.2E12;
security_level	note, warning, error, failure	
string	array of which each element is of the type character	variable VAR4: string(1 to 12):= “@$#ABC*()_%Z”;
time	an integer number of which the range is implementation defined; units can be expressed in sec, ms, us, ns, ps, fs, min and hr. . *Not supported by the Foundation synthesis program*	variable DELAY: time :=5 ns;

User-defined Types

- One can introduce new types by using the type declaration, which names the type and specifies its value range. The syntax is

  type identifier is type_definition;

- Here are a few examples of type definitions,

Integer types

type my_word_length is range 31 downto 0;

subtype data_word is my_word_length range 7 downto 0;

- A subtype is a subset of a previously defined type. The example above defines a type called **data_word** that is a sybtype of **my_word_length** of which the range is restricted from 7 to 0.
- Another example of a subtype is,

  subtype int_small is integer range -1024 to +1024;

- ***Floating-point types***

```
        type cmos_level is range 0.0 to 3.3;
        type pmos_level is range -5.0 to 0.0;
```

- ***Physical types***
- The physical type definition includes a units identifier as follows,

```
      type conductance is range 0 to 2E-9
      units
               mho;
            mmho = 1E-3 mho;                umho = 1E-6 mho;
            nmho = 1E-9 mho;                 pmho = 1E-12 mho;
       end units conductance;
```

Using subtypes in declaring objects

- Here are some object declarations that use the above types,

```
variable BUS_WIDTH: small_int :=24;
signal DATA_BUS: my_word_length;
variable VAR1: cmos_level range 0.0 to 2.5;
constant LINE_COND: conductance:= 125 umho;
```

- In order to use our own types, we need either to include the type definition inside an architecture body or to declare the type in a package. The latter can be done as follows for a package called “my_types”.

```
package my_types is
   type small_int is range 0 to 1024;
   type my_word_length is range 31 downto 0;
   subtype data_word is my_word_length is range 7 downto 0;
   -----------
```

Enumerated Types

- An enumerated type consists of lists of character literals or identifiers. The enumerated type can be very handy when writing models at an abstract level. The syntax for an enumerated type is,

  type type_name is (identifier list or character literal);
- Here are some examples,

type my_3values is (‘0’, ‘1’, ‘Z’);

type PC_OPER is (load, store, add, sub, div, mult, shiftl, shiftr);

type hex_digit is (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, 8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’);

type state_type is (S0, S1, S2, S3);

- Examples of objects that use the above types:
- ***signal SIG1: my_3values;***
- ***variable ALU_OP: pc_oper;***

- If one does not initialize the signal, the default initialization is the leftmost element of the list.
- Enumerated types have to be defined in the architecture body or inside a package.
- An example of an enumerated type that has been defined in the std_logic_1164 package is the std_ulogic type, defined as follows

```
        type STD_ULOGIC is (
                ‘U’,            -- uninitialized
                ‘X’,            -- forcing unknown
                ‘0’,            -- forcing 0
                ‘1’,            -- forcing 1
                ‘Z’,            -- high impedance
                ‘-‘);           -- don’t care
```

- In order to use this type one has to include the clause before each entity declaration.

```
        library ieee; use ieee.std_logic_1164.all;
```

Type Conversions

Conversions supported by std_logic_1164 package	
Conversion	**Function**
std_ulogic to bit	to_bit(*expression*)
std_logic_vector to bit_vector	to_bitvector(*expression*)
std_ulogic_vector to bit_vector	to_bitvector(*expression*)
bit to std_ulogic	To_StdULogic(*expression)*
bit_vector to std_logic_vector	To_StdLogicVector(*expression*)
bit_vector to std_ulogic_vector	To_StdUlogicVector(*expression*)
std_ulogic to std_logic_vector	To_StdLogicVector(*expression*)
std_logic to std_ulogic_vector	To_StdUlogicVector(*expression*)

- The IEEE std_logic_unsigned and the IEEE std_logic_arith packages allow additional conversions such as from an integer to std_logic_vector and vice versa.

- An example follows.

```
entity QUAD_NAND2 is
    port (A, B: in bit_vector(3 downto 0);
                    out4: out std_logic_vector (3 downto 0));
          end QUAD_NAND2;

architecture behavioral_2 of QUAD_NAND2 is
begin
                out4 <= to_StdLogicVector(A and B);
end behavioral_2;
```

- The expression “A and B” which is of the type **bit_vector** has to be converted to the type **std_logic_vector** to be of the same type as the output signal out4.

Operators

Class						
1. Logical operators	and	or	nand	nor	xor	xnor
2. Relational operators	=	/=	<	<=	>	>=
3. Shift operators	sll	srl	sla	sra	rol	ror
4.Addition operators	+	=	&			
5. Unary operators	+	-				
6. Multiplying op.	*	/	mod	rem		
7. Miscellaneous op.	**	abs	not			

- **Logic operators**

  X nand Y nand Z

Relational operators

Operator	**Description**	**Operand Types**	**Result Type**
=	Equality	any type	Boolean
/=	Inequality	any type	Boolean
<	Smaller than	scalar or discrete array types	Boolean
<=	Smaller than or equal	scalar or discrete array types	Boolean
>	Greater than	scalar or discrete array types	Boolean
>=	Greater than or equal	scalar or discrete array types	Boolean

Example

```
variable STS               : Boolean;
 constant A                 : integer :=24;
constant B_COUNT   : integer :=32;
constant C                 : integer :=14;
STS <= (A < B_COUNT) ;  -- will assign the value “TRUE” to STS
STS <=  ((A >= B_COUNT) or (A > C));    -- will result in “TRUE”
STS <=  (std_logic (‘1’, ‘0’, ‘1’) < std_logic(‘0’, ‘1’,’1’));
                                                    --makes STS “FALSE”
type new_std_logic is (‘0’, ‘1’, ‘Z’, ‘-‘);
variable A1: new_std_logic :=’1’;
variable A2: new_std_logic :=’Z’;
STS <=  (A1 < A2);
          ---will result in “TRUE” since ‘1’ occurs to the left of ‘Z’.
```

- **<u>Shift operators</u>**
- These operators perform a bit-wise shift or rotate operation on a one-dimensional array of elements of the type bit (or std_logic) or Boolean.
- The operand is on the left of the operator and the number (integer) of shifts is on the right side of the operator. As an example,

```
        variable NUM1          :bit_vector := “10010110”;
        NUM1 srl 2;
```

- will result in the number “00100101”.

Operator	Description	Operand Type	Result Type
sll	Shift left logical (fill right vacated bits with the 0)	Left: Any one-dimensional array type with elements of type bit or Boolean; Right: integer	Same as left type
srl	Shift right logical (fill left vacated bits with 0)	same as above	Same as left type
sla	Shift left arithmetic (fill right vacated bits with rightmost bit)	same as above	Same as left type
sra	Shift right arithmetic (fill left vacated bits with leftmost bit)	same as above	Same as left type
rol	Rotate left (circular)	same as above	Same as left type
ror	Rotate right (circular)	same as above	Same as left type

Behavioral Modeling: Sequential Statements

- **In this section we will discuss different constructs for describing the behavior of components and circuits in terms of sequential statements.**
- **The basis for sequential modeling is the *process* construct.**
- **The *process* construct allows us to model complex digital systems, in particular sequential circuits.**

a. Process

- **A process statement is the main construct in behavioral modeling that allows you to use sequential statements to describe the behavior of a system over time.**

The syntax for a process statement

```
[process_label:] process [ (sensitivity_list) ] [is]
                   [ process_declarations]
          begin
                   list of sequential statements such as:
                            signal assignments
                            variable assignments
                            case statement
                            exit statement
                            if statement
                            loop statement
                            next statement
                            null statement
                            procedure call
                            wait statement
          end process [process_label];
```

Positive edge-triggered D flip-flop with asynchronous clear input (sequential circuit)

```
entity DFF_CLEAR is
    port (CLK, CLEAR, D : in std_logic;
        Q : out std_logic);
end DFF_CLEAR;
architecture BEHAV_DFF of DFF_CLEAR is
Begin
DFF_PROCESS: process (CLK, CLEAR)
    begin
        if (CLEAR = ‘1’) then
            Q <= ‘0’;
        elsif (CLK’event and CLK = ‘1’) then
            Q <= D;
        end if;
    end process;
end BEHAV_DFF;
```

Process label

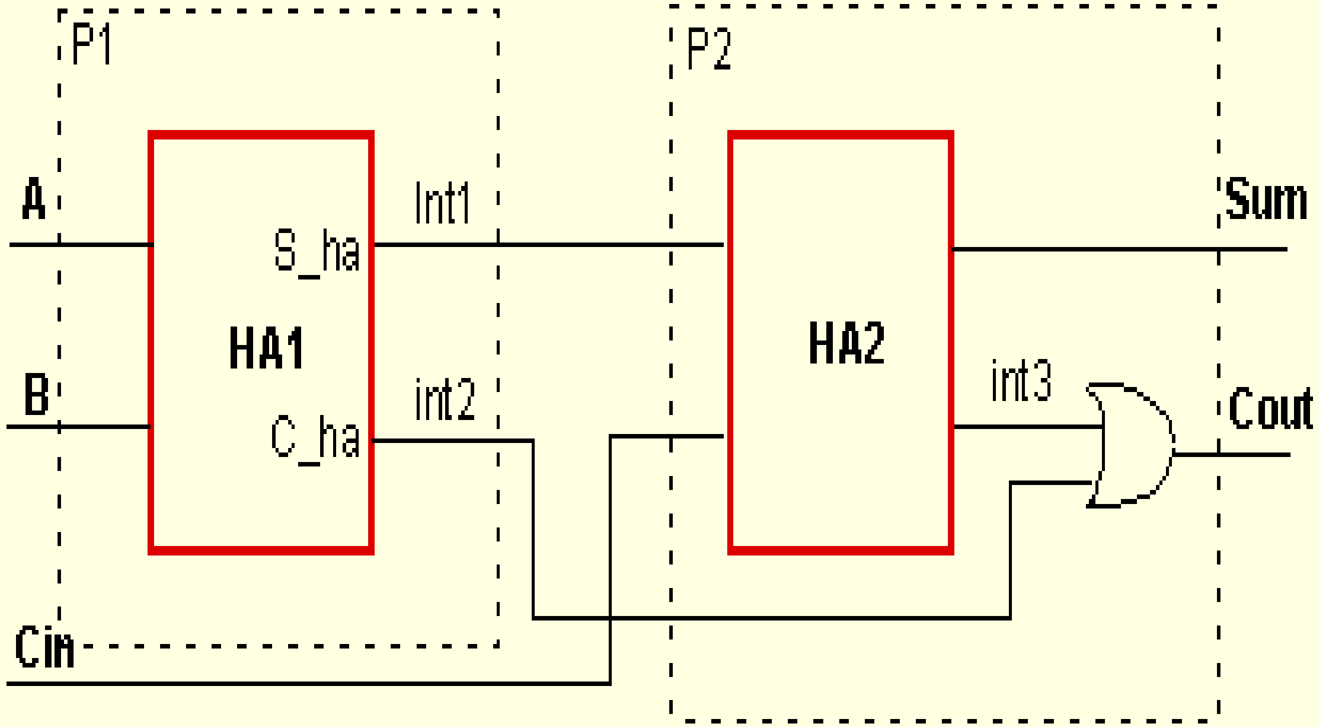
Sensitivity list

Any **change** in the **value of the signals** in the sensitivity list will **cause immediate execution** of the process.

signal_name’event returns the Boolean value True if an event on the signal occurred, otherwise gives a False

checks for a positive clock edge (clock event (change) AND clock is high).

Example for a Full Adder, composed of two Half Adders (combinational circuit)

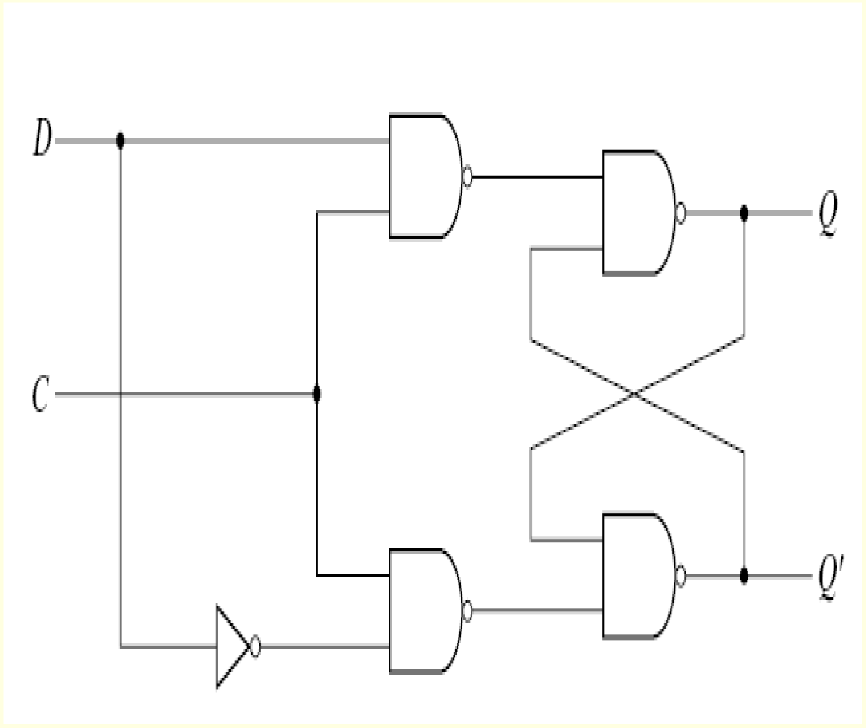


Example for a Full Adder, composed of two Half Adders (combinational circuit)

```
entity FULL_ADDER is
     port (A, B, Cin : in std_logic;
           Sum, Cout : out std_logic);
end FULL_ADDER;
architecture BEHAV_FA of FULL_ADDER is
signal int1, int2, int3: std_logic;
begin
                    -- Process P1 that defines the first half adder
P1: process (A, B)
     begin
           int1<= A xor B;
           int2<= A and B;
     end process;
     -- Process P2 that defines the second half adder and the OR -- gate
P2: process (int1, int2, Cin)
     begin
          Sum <= int1 xor Cin;
           int3 <= int1 and Cin;
           Cout <= int2 or int3;
     end process;
end BEHAV_FA;
```

Example: D-flip flop

```
entity DFF1 is
   Port ( D : in std_logic;
          CLK : in std_logic;
          Q : out std_logic;
          QN : out std_logic);
end DFF1;
architecture Behavioral of DFF1 is
begin
           process (CLK)
           begin
       if (CLK = '1' and CLK'event)  then
                       Q <= D;
                       QN <= NOT D;
                       end if;
         end process;
end Behavioral;
```



b. If Statements

- The if statement executes a sequence of statements whose sequence depends on one or more conditions. The syntax is as follows:

```
    if condition then
          sequential statements
    [elsif condition then
          sequential statements ]
    [else
          sequential statements ]
  end if;
```

Example

```
IF (day = sunday) THEN
   weekend := TRUE;
ELSIF (day = saturday) THEN
   weekend := TRUE;
ELSE
   weekday := TRUE;
END IF;
```

Example for a 4-to-1 multiplexer with inputs A, B, C and D, and select signals S0 and S1.

```
entity MUX_4_1a is
  port (S1, S0, A, B, C, D: in std_logic;
        Z: out std_logic);
  end MUX_4_1a;
architecture behav_MUX41a of MUX_4_1a is
  begin
P1: process (S1, S0, A, B, C, D)
       begin
     if S1='0' and  S0='0' then
        Z <= A;
     elsif S1='0' and  S0='1' then
        Z <= B;
     elsif S1='1' and  S0='0' then
            Z <= C;
     elsif S1='1' and  S0='1' then
            Z <= D;
            end if;
  end process P1;
end behav_MUX41a;
```

c. Case statements

- The case statement executes one of several sequences of statements, based on the value of a single expression. The syntax is as follows,

```
    case expression is
      when choices =>
              sequential statements
      when choices =>
              sequential statements
           -- branches are allowed
      [ when others => sequential statements ]
  end case;
```

c. Case statements

- The expression must evaluate to an integer, an enumerated type of a one-dimensional array, such as a bit_vector.
- The case statement evaluates the expression and compares the value to each of the choices.
- The when clause corresponding to the matching choice will have its statements executed.

The following rules must be adhered to:

- no two choices can overlap (i.e. each choice can be covered only once)
- if the “when others” choice is not present, all possible values of the expression must be covered by the set of choices.

```
entity GRD_201 is
    port(VALUE: in integer range 0 to
    100;
         A, B, C, D,F: out bit);
end GRD_201;
architecture behav_grd of GRD_201 is
begin
    process (VALUE)
      A <= '0';
      B <= '0';
      C <= '0';
      D <= '0';
      F <= '0';
      begin
case VALUE is
            when 51 to 60 =>
                D <= '1';
            when 61 to 70 | 71 to 75 =>
                C <= '1';
            when 76 to 85 =>
                B <= '1';
             when 86 to 100 =>
                A <= '1';
             when others  =>
               F <= '1';
          end case;
      end process;
end behav_grd;
```

Example using the case construct: 4-to-1 MUX.

```
entity MUX_4_1 is
   port ( SEL: in std_logic_vector(2 downto 1);
          A, B, C, D: in std_logic;
          Z: out std_logic);
   end MUX_4_1;
architecture behav_MUX41 of MUX_4_1 is
begin
   PR_MUX: process (SEL, A, B, C, D)
   begin
     case SEL is
          when “00” => Z <= A;
          when “01” => Z <= B;
          when “10” => Z <= C;
          when “11” => Z <= D;
          when others => Z <= ‘X’;
     end case;
   end process PR_MUX;
end behav_MUX41;
```

```
CASE instruction IS
WHEN load_accum => accum <= data;
WHEN store_accum => data_out <= accum;
WHEN load|store => process_IO(addr);
WHEN OTHERS =>
process_error(instruction);
END CASE;
```

The null statement

```
entity EX_WAIT is
  port ( CNTL: in integer range 0 to 31;
         A, B: in std_logic_vector(7 downto 0);
         Z: out std_logic_vector(7 downto 0) );
  end EX_WAIT;
architecture arch_wait of EX_WAIT is
begin
  P_WAIT: process (CNTL)
  begin
   Z <=A;
   case CNTL is
       when 3 | 15 =>
              Z <= A xor B;
       when others =>
              null;
   end case;
end process P_WAIT;
end arch_wait;
```

When the value of CNTL is 3 or 15, the signals A and B will be xor-ed

The null statement states that no action will occur.

Wait statement

- The wait statement will halt a process until an event occurs. There are several forms of the wait statement,

The condition in the “wait until” statement must be TRUE for the process to resume.

```
wait until condition;
wait for time expression;
wait on signal;
wait;
```

Give a specific time for wait

Waits for a signal to be=1

Wait statement

- The syntax is as follows,

```
wait until signal = value;
wait until signal'event and signal = value;
wait until not signal'stable and signal =
        value;
```

- The condition in the "wait until" statement must be TRUE for the process to resume.
- A few examples follow.

```
wait until CLK='1';
wait until CLK='0';
wait until CLK'event and CLK='1';
wait until not CLK'stable and CLK='1';
```

WAIT ON a, b;

- When an event occurs on either **a or b, the process resumes with the** statement following the **WAIT statement.**

WAIT UNTIL ((x * 10) < 100);

- In this example, as long as the value of signal **x is greater than or equal** to 10, the **WAIT statement suspends the process or subprogram.**
- **When the** value of **x is less than 10, execution continues with the statement following** the **WAIT statement.**

WAIT FOR 10 ns; **WAIT FOR (a * (b + c));**

- In the first example, the time expression is a simple constant value.
- The **WAIT statement suspends execution for 10 nanoseconds.**
- **After 10** nanoseconds has elapsed, execution continues with the statement following the **WAIT statement.**

```
PROCESS
BEGIN
    WAIT UNTIL clock = ‘1’ AND clock’EVENT;
    q <= d;
END PROCESS;
```

```
PROCESS
BEGIN
    WAIT UNTIL clock = ‘1’ AND clock’EVENT;
    IF (reset = ‘1’) THEN
           q <= ‘0’;
    ELSE
           q <= d;
    END IF;
END PROCESS;
```

Multiple WAIT Conditions

- A single statement can include an **ON signal, UNTIL** expression, and **FOR time_expression clauses.**

WAIT ON nmi,interrupt UNTIL ((nmi = TRUE) or

(interrupt = TRUE)) FOR 5 usec;

- This statement waits for an event on signals **nmi and interrupt and** continues only if **interrupt or nmi is true at the time of the event, or until** 5 msec of time has elapsed.
- Only when one or more of these conditions are true does execution continue.

WAIT UNTIL (interrupt = TRUE) OR (old_clk = ‘1’);

- Be sure to have at least one of the values in the expression contain a signal.

- This is necessary to ensure that the **WAIT statement does not wait forever.**
- If both **interrupt and old_clk are variables, the WAIT statement does not** reevaluate when these two variables change value.
- Only signals have events on them, and only signals can cause a **WAIT** statement or concurrent signal assignment to reevaluate.

Loop statements

- A loop statement is used to repeatedly execute a sequence of sequential statements. The syntax for a loop is as follows:

```
[ loop_label :]iteration_scheme loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
  end loop [loop_label];
```

- The next statement terminates the rest of the current loop iteration and execution will proceed to the next loop iteration.
- The exit statement skips the rest of the statements, terminating the loop entirely, and continues with the next statement after the exited loop.

the loop contains no wait statements

Basic Loop statement

- There are three types of iteration schemes:
  - basic loop
  - while … loop
  - for … loop
- This loop has no iteration scheme. It will be executed continuously until it encounters an exit or next statement.

```
[ loop_label :] loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
    end loop [ loop_label];
```

Example of a basic loop to implement a counter that counts from 0 to 31

```
entity COUNT31 is
  port ( CLK: in std_logic;
          COUNT: out integer);
  end COUNT31;
architecture behav_COUNT of COUNT31 is
begin
P_COUNT: process
    variable intern_value: integer :=0;
  begin
    COUNT <= intern_value;
    loop
      wait until CLK='1';
          intern_value:=(intern_value + 1) mod 32;
          COUNT <= intern_value;
    end loop;
end process P_COUNT;
end behav_COUNT;
```

Internal variable to hold count before sending it out

Wait until it is executed to go to the next line

For-Loop statement

- The for-loop uses an integer iteration scheme that determines the number of iterations.
- The syntax is as follows:

```
[ loop_label :] for identifier in range loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
   end loop[ loop_label ];
```

- The identifier (index) is automatically declared by the loop itself, so one does not need to declare it separately.
- The value of the identifier can only be read inside the loop and is not available outside its loop.
- One cannot assign or change the value of the index.
- This is in contrast to the while-loop whose condition can involve variables that are modified inside the loop.

```
process (A)
begin
Z <= "0000";
for i in 0 to 3 loop
if (A = i) then Z(i) <= '1';
end if;
 end loop;
end process;
```

```
for I in A'low to A'high loop
 TMP := TMP xor A(I);
end loop;
```

```
for i in 1 to 10 loop
 if (REPEAT = '1') then i := i-1; -- Illegal
end if;
end loop;
```

- The values used to specify the range in the **FOR loop need not be specific** integer values, as has been shown in the examples. The range can be any discrete range. A **discrete_range can be expressed as a subtype_indication or a range statement.**

```
PROCESS(clk)
    TYPE day_of_week IS (sun, mon, tue, wed, thur, fri,sat);
BEGIN
    FOR i IN day_of_week LOOP
           IF i = sat THEN
                      son <= mow_lawn;
           ELSIF i = fri THEN
                      visit<= family;
           ELSE
                      dad <= go_to_work;
           END IF;
    END LOOP;
END PROCESS;
```

- In this example, the range is specified by the type. By specifying the type as the range, the compiler determines that the leftmost value is **sun,** and the rightmost value is **sat. The range then is determined as from sun to sat.**

- If an ascending range is desired, use the **to clause. The downto clause** can be used to create a descending range.

```
PROCESS(x, y)
BEGIN
    FOR i IN x downto y LOOP
        q(i) := w(i);
    END LOOP;
END PROCESS;
```

```
FOR i IN 1 to 10 LOOP
    i_squared(i) := i * i;
END L

PROCESS(i)
    BEGIN
    x <= i + 1; -- x is a signal
    FOR i IN 1 to a/2 LOOP
          q(i) := a; -- q is a variable
    END LOOP;
END PROCESS;
```

While-Loop statement

- The while … loop evaluates a Boolean iteration condition.
- When the condition is TRUE, the loop repeats, otherwise the loop is skipped and the execution will halt.
- The syntax for the while…loop is as follows,

```
  [ loop_label :] while condition loop
        sequential statements
        [next  [label] [when
condition];
        [exit  [label] [when condition];
    end loop[ loop_label ];
```

- The condition of the loop is tested before each iteration, including the first iteration.
- If it is false, the loop is terminated.

```
process (A)
variable I : integer range 0 to 4;
Begin
 Z <= "0000";
i := 0;
while (i <= 3) loop
 if (A = i) then Z(i) <= '1';
end if;
i := i + 1;
end loop;
end process;
```

Next and Exit Statement

- The next statement skips execution to the next iteration of a loop statement and proceeds with the next iteration.
- The syntax is

```
next [label] [when condition];
```

- The when keyword is optional and will execute the next statement when its condition evaluates to the Boolean value TRUE.

Next and Exit Statement

- The exit statement skips the rest of the statements, terminating the loop entirely, and continues with the next statement after the exited loop.
- The syntax is as follows:

  exit [label] [when condition];

- The when keyword is optional and will execute the next statement when its condition evaluates to the Boolean value TRUE.
- Notice that the difference between the next and exit statement, is that the exit statement terminates the loop.

Dataflow Modeling – Concurrent Statements

- Behavioral modeling can be done with *sequential* statements using the process construct or with concurrent statements.
- The first method was described in the previous section and is useful to describe complex digital systems.
- The *concurrent* statements are used to describe behavior.
- This method is usually called dataflow modeling.
- The dataflow modeling describes a circuit in terms of its *function* and the *flow of data* through the circuit.
- This is different from the *structural* modeling that describes a circuit in terms of the interconnection of components.
- Concurrent signal assignments are event triggered and executed as soon as an event on one of the signals occurs.

WITH-SELECT

- The with-select statement is used for selective signal assignment.
- It is a concurrent statement.

Syntax

```
  with  expression  select:
 target    <=  expression 1  when  choice1
               expression 2  when  choice2
                           .
                           .
                           .
              expression N  when  others;
```

Example:

```
entity mux2 is
      port ( i0, i1 : in bit_vector(1 downto 0);
                  y  : out bit_vector(1 downto 0);
                 sel : in bit
              );
end mux2;

architecture behaviour of mux2 is
begin
      with sel select
                y <= i0 when '0',
                i1 when '1';
end behaviour;
```

WHEN-ELSE

syntax :

```
Signal_name<= expression1  when  condition1  else
              expression2  when  condition2  else
              expression3;
```

Example:

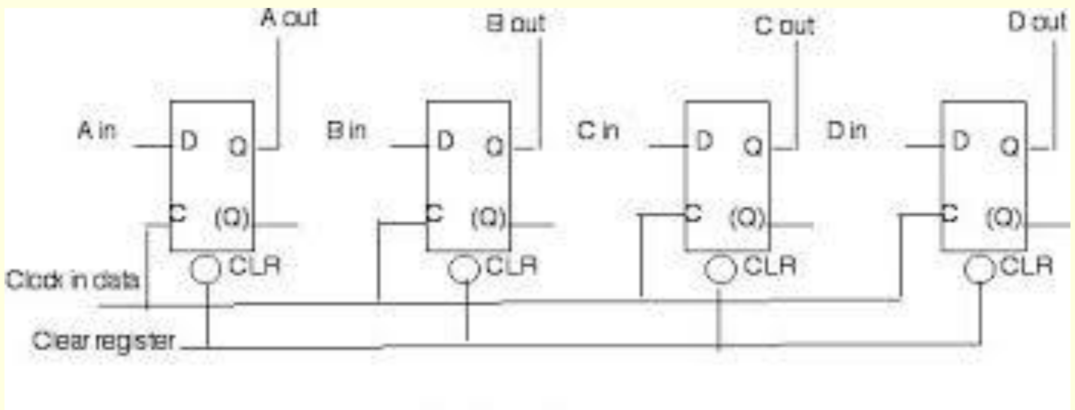
```
entity tri_state is
   port (a, enable : in std-logic;
            b : out std_logic);
end tri_state;
architecture beh of tri_state is
begin
      b <=  a when enable ='1' else
            'Z';
end beh;
```

WHEN-ELSE VS. WITH-SELECT

- In the ‘with’ statement,choice is limited to the choices provided by the with ‘express-ion’.
- In the ‘when’ statement each choice itself can be a separate expression.
- when statement is prioritized (since each choice can be a different expression,more than one condition can be true at the same time, thus necessitating a priority based assignment)
- with statement does not have any priority (since choices are mutually exclusive) .

Registers

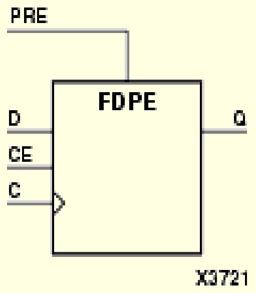
- Registers are groups of flip-flops, where each flip-flop is capable of storing one bit of information.
- An *n*-bit register is a group of *n* flip-flops.
- The basic function of a register is to hold information in a digital system and make it available to the logic elements for the computing process.
- Registers consist of a finite number of flip-flops. Since each flip-flop is capable of storing either a "0" or a "1", there is a finite number of 0-1 combinations that can be stored into a register. Each of those combinations is known as *state* or *content* of the register.



(presets have been omitted for clarity)

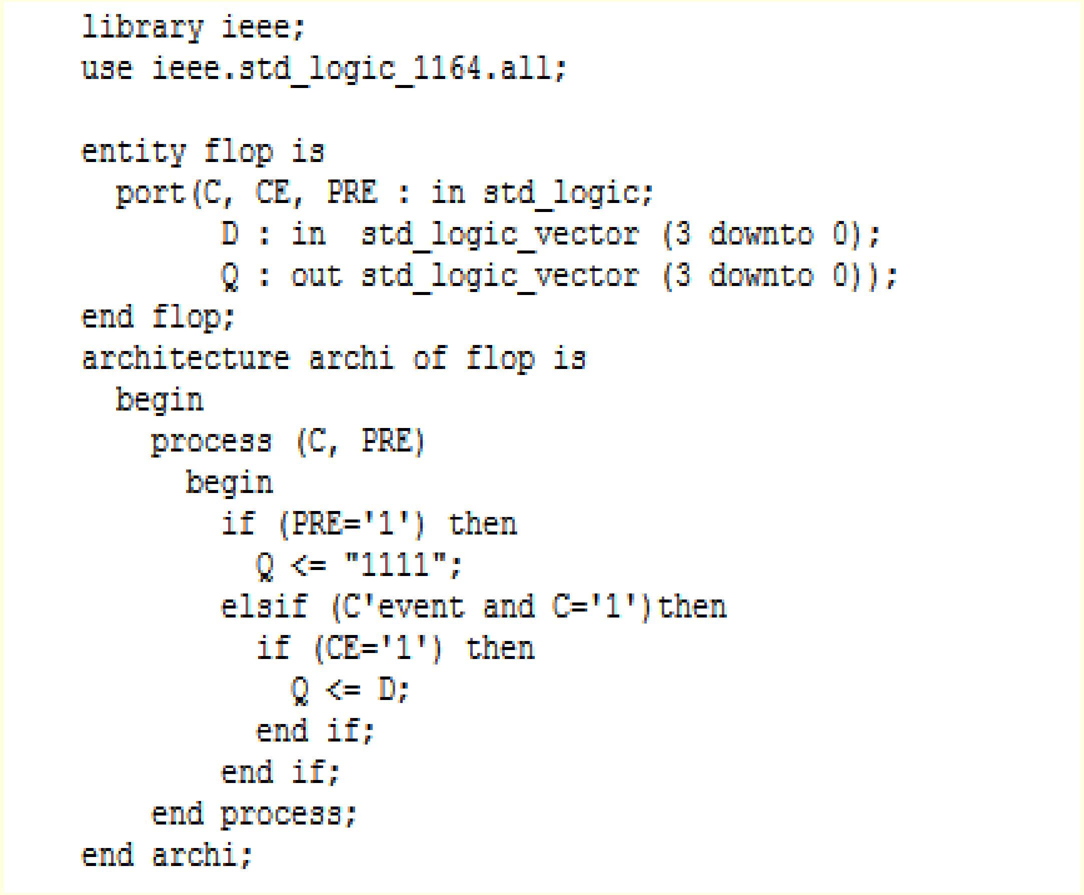
4-bit Register with positive-edge clock, asynchronous set and clock enable

The following table shows pin definitions for a 4-bit register with positive-edge clock, asynchronous set and clock enable



IO Pins	Description
D[3:0]	Data Input
C	Positive-Edge Clock
PRE	Asynchronous Set (active High)
CE	Clock Enable (active High)
Q[3:0]	Data Output

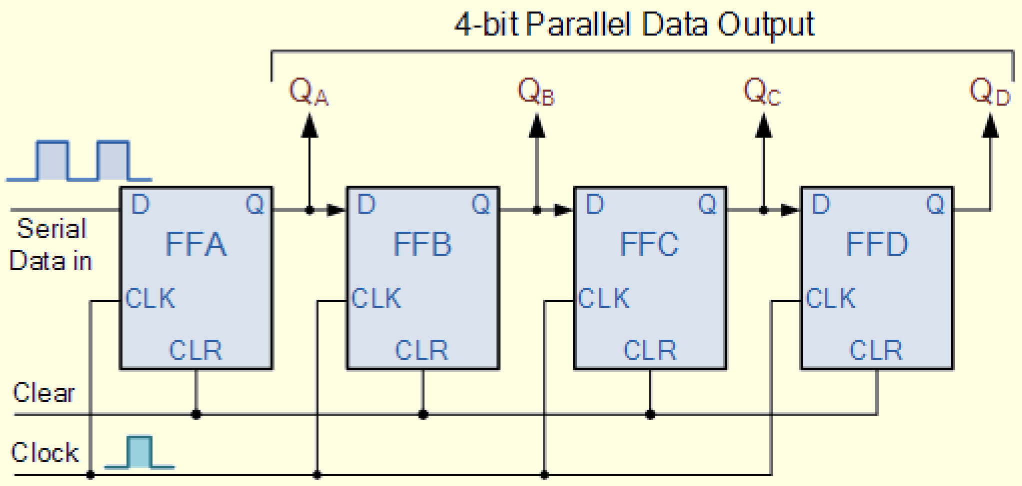
4-bit Register with positive-edge clock, asynchronous set and clock enable



```
library ieee;
use ieee.std_logic_1164.all;

entity flop is
  port(C, CE, PRE : in std_logic;
        D : in  std_logic_vector (3 downto 0);
        Q : out std_logic_vector (3 downto 0));
end flop;
architecture archi of flop is
  begin
    process (C, PRE)
      begin
        if (PRE='1') then
          Q <= "1111";
        elsif (C'event and C='1')then
          if (CE='1') then
            Q <= D;
          end if;
        end if;
    end process;
end archi;
```

Shift Registers

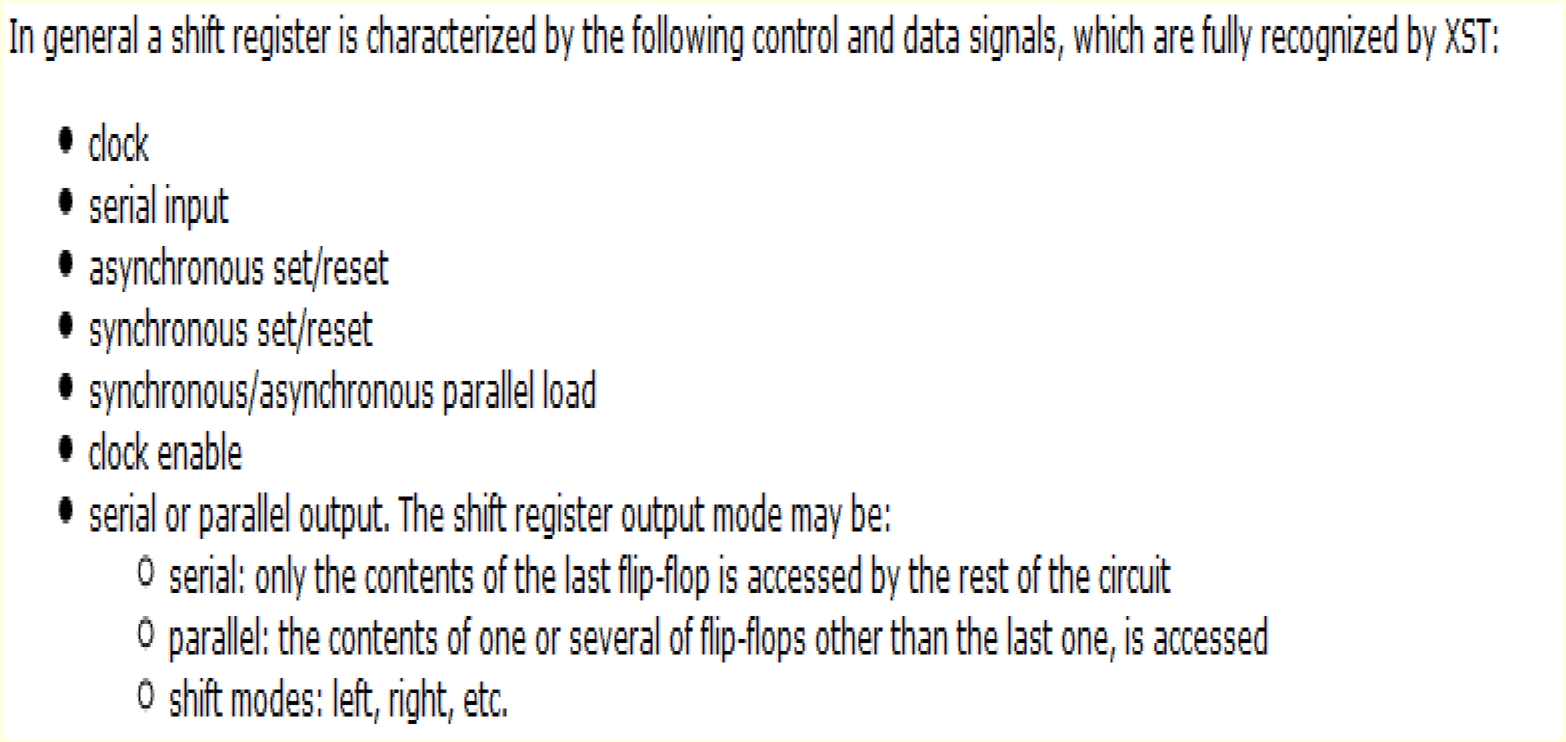
- Information often comes bitwise i.e. one bit at every clock pulse. Shift registers are used to store such data.
- A shift register has one serial input. Every clock pulse one bit is loaded from serial in into the first flip-flop of the register while all the actual flip-flop contents are shifted to the next flip-flop, dropping the last bit.
- Shift registers may feature a serial output so that the last bit that gets shifted out of the register can be processed further.
- It is thus possible to build up a chain of shift registers by connecting each serial out to another shift register's serial in, effectively creating a single big shift register.
- It is also possible to create a Cyclic register by connecting the serial out to the same register's serial in.



Shift Registers-2

- Shift register circuits may also feature additional parallel-in functionality that allows manipulation of individual bits. If the output of all flip-flops (and therefore the register's complete content) are read from the lines Q1 to Qn the register is used as Serial In – Parallel Out (SIPO).
- A typical purpose for such a SIPO register is to collect data that is delivered bitwise and that is needed in n-bit data words (e.g. to convert the signals from serial ports of a computer: the line transports 1 bit a time, the computer uses 8, 16 or 32 bit datawords).
- Shifting bits are important for mathematical operations: if the output of the whole register is interpreted as a binary number, shifting by one bit corresponds to multiplying or dividing by 2 (depends on which flip-flop is interpreted as MSB).there are 4 types of shift registers-Serial In-parallel Out(SIPO),Serial In-Serial Out(SISO),Parallel In-Serial Out(PISO),Parallel In-Parallel Out(PIPO)

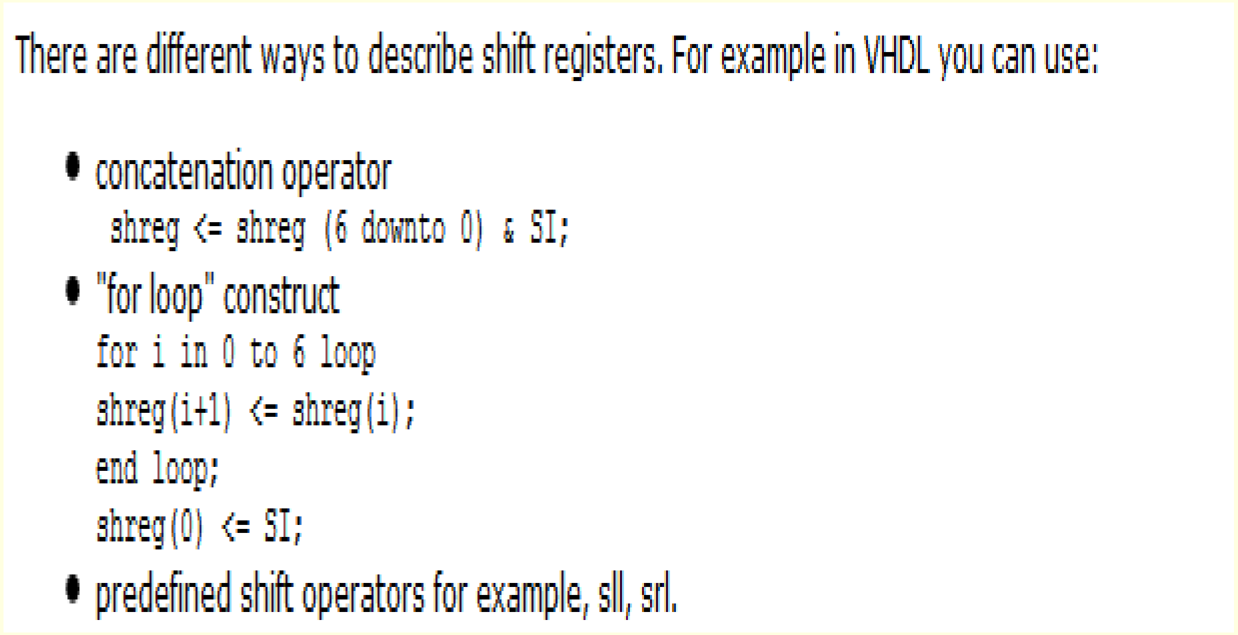
Shift Registers-3



In general a shift register is characterized by the following control and data signals, which are fully recognized by XST:

- clock
- serial input
- asynchronous set/reset
- synchronous set/reset
- synchronous/asynchronous parallel load
- clock enable
- serial or parallel output. The shift register output mode may be:
    - serial: only the contents of the last flip-flop is accessed by the rest of the circuit
    - parallel: the contents of one or several of flip-flops other than the last one, is accessed
    - shift modes: left, right, etc.

Shift Registers-4



There are different ways to describe shift registers. For example in VHDL you can use:

- concatenation operator

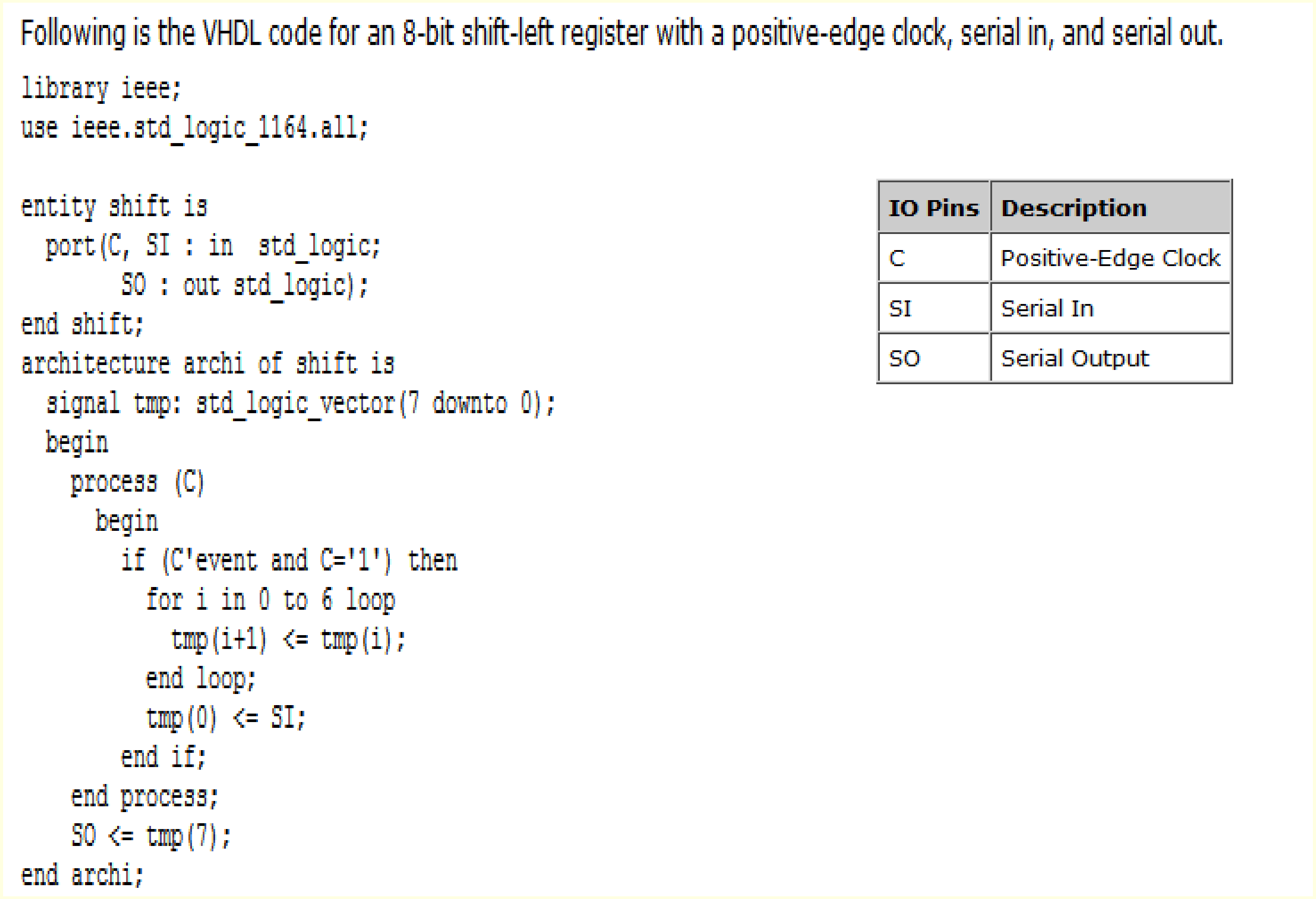
```
 shreg <= shreg (6 downto 0) & SI;
```

- "for loop" construct

```
for i in 0 to 6 loop
shreg(i+1) <= shreg(i);
end loop;
shreg(0) <= SI;
```

- predefined shift operators for example, sll, srl.

8-bit Shift-Left register with a positive-edge clock, serial in, serial out

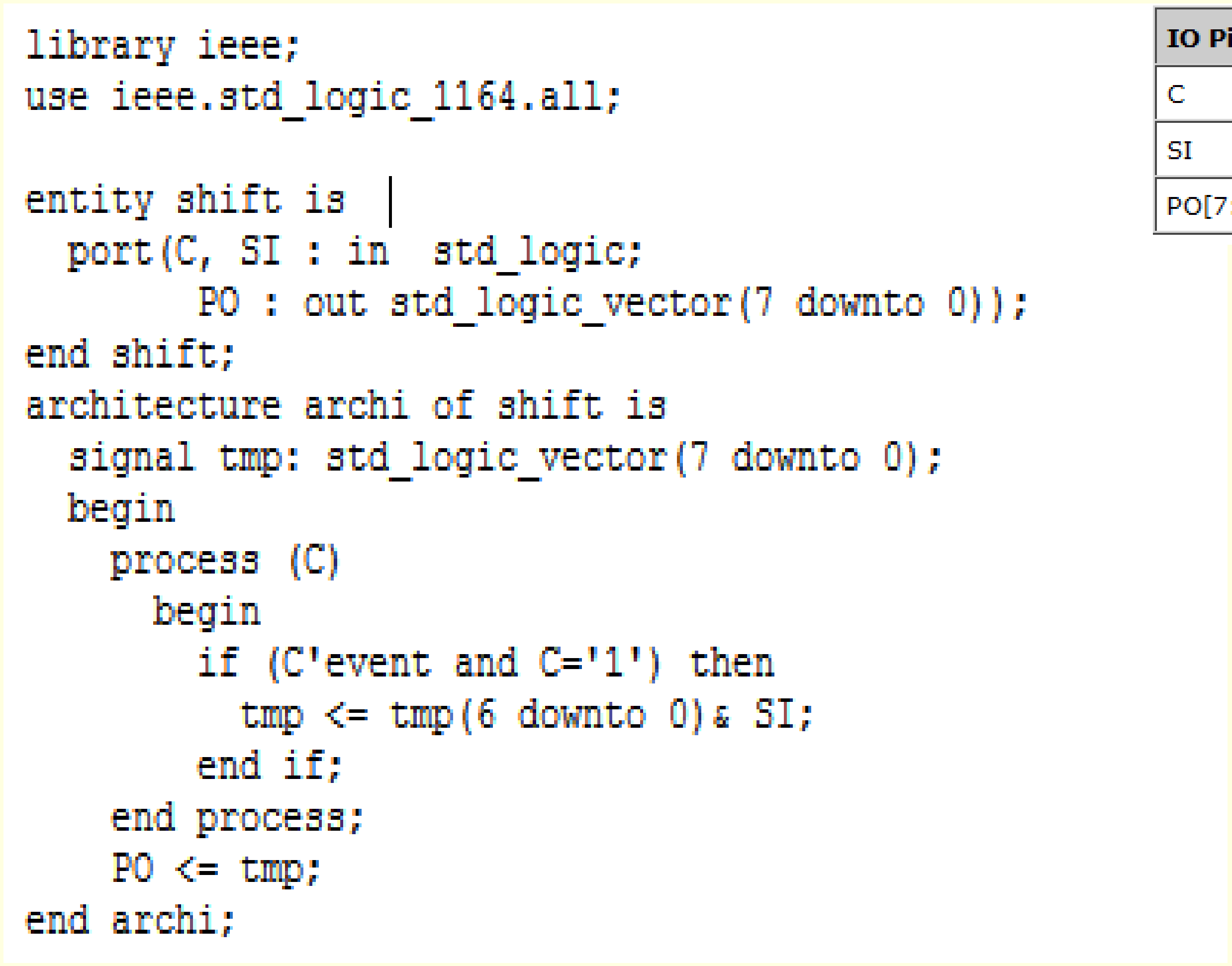


Following is the VHDL code for an 8-bit shift-left register with a positive-edge clock, serial in, and serial out.

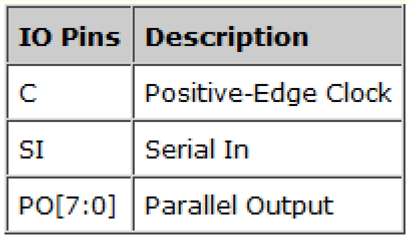
```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(C, SI : in  std_logic;
       SO : out std_logic);
end shift;
architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
      begin
        if (C'event and C='1') then
          for i in 0 to 6 loop
            tmp(i+1) <= tmp(i);
          end loop;
          tmp(0) <= SI;
        end if;
    end process;
    SO <= tmp(7);
end archi;
```

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
SO	Serial Output

8-bit Shift-Left Register with Positive-Edge Clock, Serial In, and Parallel Out

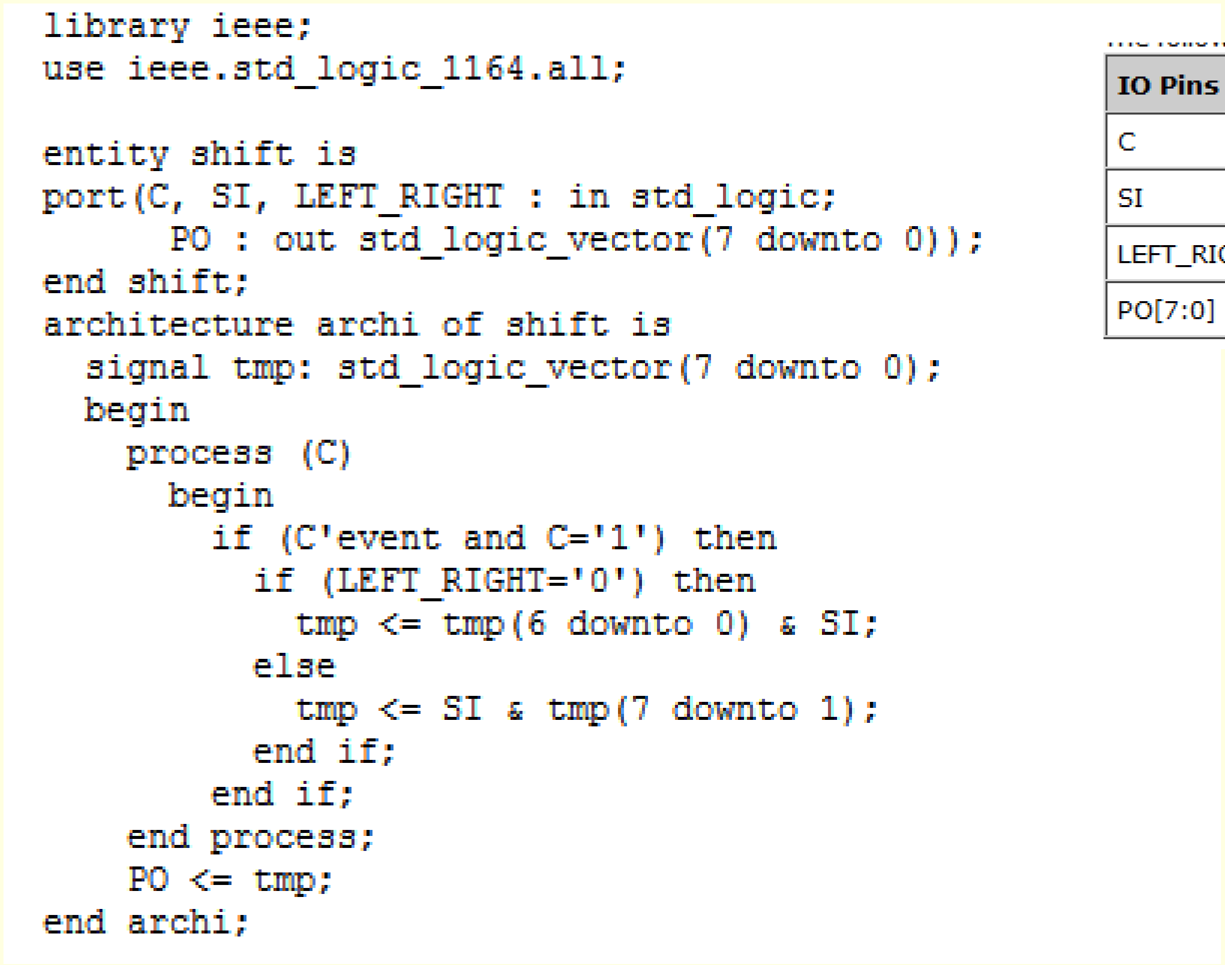


```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(C, SI : in  std_logic;
        PO : out std_logic_vector(7 downto 0));
end shift;
architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
      begin
        if (C'event and C='1') then
          tmp <= tmp(6 downto 0)& SI;
        end if;
    end process;
    PO <= tmp;
end archi;
```

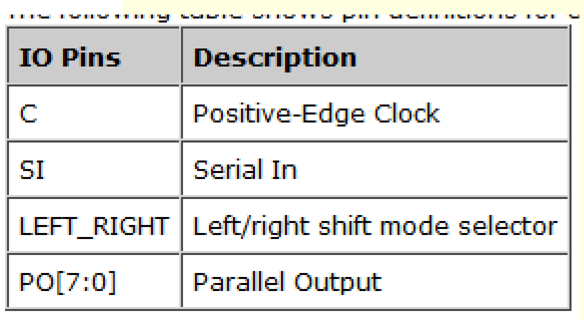


IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
PO[7:0]	Parallel Output

8-bit Shift-Left/Shift-Right Register with Positive-Edge Clock, Serial In, and Parallel Out



```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
port(C, SI, LEFT_RIGHT : in std_logic;
      PO : out std_logic_vector(7 downto 0));
end shift;
architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
      begin
        if (C'event and C='1') then
          if (LEFT_RIGHT='0') then
            tmp <= tmp(6 downto 0) & SI;
          else
            tmp <= SI & tmp(7 downto 1);
          end if;
        end if;
    end process;
    PO <= tmp;
end archi;
```



IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
LEFT_RIGHT	Left/right shift mode selector
PO[7:0]	Parallel Output

With our best wishes